Typescript-on-LLVM
Language Reference Manual

Ratheet Pandya
UNI: rp2707
COMS 4115 H01 (CVN)

1. Introduction

This manual describes the Typescript-on-LLVM Language, as described in the project proposal.

In the following I borrow heavily from the language, format, and structure used in *The C Programming Language, Second Edition* (Kernighan and Ritchie, 1988) since many of same syntactic and semantic rules for C apply to Typescript-on-LLVM.

2. Lexical Conventions

2.1 Tokens

There are six kinds of tokens:

- Identifiers
- Keywords
- String literals
- Operators
- Other separators

Whitespace (defined here) and comments (described below) are ignored except as they separate tokens:

- Blanks
- Horizontal and vertical tabs
- Newlines

Some whitespace is required in order to separate otherwise adjacent identifiers, keywords, and constants. If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

2.2 Comments

Comments begin with the characters `/*` and end with the characters `*/`. Comments do not nest and do not occur within string literals.

2.3 Identifiers

An identifier (or name) is a sequence of letters and digits with the following properties:

- The first character must be a letter or an underscore (‘_’).
- Identifiers are case-sensitive.
- Identifiers may have any length.

Generally, identifiers are names that are bound to functions or variables in a scoped namespace, as explained below.

2.4 Reserved Keywords

The following identifiers are used as keywords, and may not be used otherwise:

```
and
boolean
const
else
false
for
function
if
let
null
number
or
return
true
string
void
while
```

2.5 String Literals

A String Literal is a sequence of one or more characters enclosed in double-quotes, e.g. “foo”. In order to include a double-quote in a string constant, the escape character slash (‘\’) may be used. To include a slash in a string, use another slash, e.g. “\\x” may be used to represent the literal string “\x”.

2.6 Operators

Typescript-on-LLVM only supports binary operators. The table below summarizes the operators supported:

Operator	Use
+	Arithmetic addition
-	Arithmetic subtraction
*	Arithmetic multiplication
/	Arithmetic division
=	Assignment
!=	Not-equal boolean comparison
<	Less-than boolean comparison
<=	Less-than-or-equal-to boolean comparison
>	Greater-than boolean comparison
>=	Greater-than-or-equal-to boolean comparison
==	Equals boolean comparison

2.6.1 Associativity

The assignment operator, '=', is right-associative. The remaining operators are left associative, with the following rules:

- = takes precedence over !=
- The following operators are in precedence order, left-to-right: < > <= >=
- + takes precedence over -
- * takes precedence over /

2.7 Other Separators

The other separator tokens include:

- the semicolon '`;`' for sequencing of statements
- the comma, ',' for separating items in an array
- curly braces, '`{`' and '`}`' for block-scoping
- parentheses, '`(`' and '`)`' for parameter lists in function definitions

3. Expressions

Expressions refer to program code that may be evaluated in a particular scope. There are two two kinds of top-level expressions: **primary** and **postfix**.

Primary expressions have the following form:

primary-expression:
  identifier
  boolean-value
  number-value
  string-value

Postfix expressions group operators left-to-right:

postfix-expression:
  primary-expression
  postfix-expression `[` *expression* `]`
  postfix-expression `(` *argument-expression-list*$_{opt}$ `)`

argument-expression-list:
  assignment-expression-list
  argument-expression-list `,` *assignment-expression*

Assignment expressions are covered in Section 3.1.

As shown here, array indexing is done via a postfix expression in which the expression inside brackets resolves to the integer index of the array.

Similarly, function calls are postfix expressions that contain zero or more argument expressions within parentheses.

3.1 Operator Expressions

For the non-assignment operators described in Section 2.6, the following forms are used:

binary-expression:
expression `+` *expression*
expression `-` *expression*
expression `*` *expression*
expression `/` *expression*

conditional-expression:
expression `<` *expression*
expression `>` *expression*
expression `<=` *expression*
expression `>=` *expression*
expression `==` *expression*
expression `and` *expression*
expression `or` *expression*

Assignments take the following form:

assignment-expression:
identifier `=` *primary-expression* `;`
identifier `:` *type-specifier* `=` *primary-expression* `;`

Assignments are described in more detail in Section 5.

3. Statements

Statements are sequences of expressions, and have the following form:

statement:
expression-statement
compound-statement
conditional-statement
loop-statement

expression-statement:

expression$_{opt}$;

compound-statement:
`{` *statement-list*$_{opt}$ `}`

statement-list:
statement
statement-list statement

Conditional and loop statements are defined in Section 3.1 and Section 3.2, respectively.

3.1 Conditionals

Conditional statements allow for flow control based on the boolean value of a given conditional expression:

conditional-statement:
`if (` *conditional-expression* `)` *compound-statement*
`if (` *conditional-expression* `)` *compound-statement* `else` *compound-statement*

For example:

```
if (x < y) {
  print(x);
}

if (x > y) {
  print(y);
} else {
  print(x);
}
```

3.2 Looping

Looping is supported using the `while` construct, and has the following form:

loop-statement:
while `(` *conditional-expression* `)` *compound-statement*

For example:

```
while (x < 100) {
  printAndIncrement(x);
}
```

4. Functions

Functions are optionally parameterized scoped blocks of code that are given identifiers in the global namespace.

4.1 Function Definition

Function definitions have the form `function D : T` *compound-statement*, where

- `D` has the form `D'(` *parameter-list* `)`
    - `D'` denotes the identifier for the function
- `T` denotes the return type of the function

The syntax of parameters is:

parameter-list:
  parameter-list
  parameter-list , parameter-declaration

parameter-declaration:
  identifier `:` *type-specifier*

For example:

```
function add(x : number, y : number) {
    return x + y;
}
```

Here:

- `add` corresponds to D' above
- `x : number, y : number` is the *parameter-list*
- { return x + y; } is the *compound-statement*

5. Variables and Constants

Variables and constants are bindings of expressions that may be assigned to identifiers in a scoped namespace.

5.2.1 The `let` keyword

The `let` keyword is used to define a scope for an expression and assign that expression to a variable.

`let` assignment has the following form:

let-expression:
  `let` *assignment-expression*

For example:

```
let s = "Hello world!";
```

5.2.2 The `const` keyword

A constant is an immutable binding of an identifier to a value. Constants are declared using the `const` keyword, which binds an expression to a constant (non-reassignable) identifier.

const assignment has the following form:

const-expression:
  `const` *assignment-expression*

For example:

```
const s = "Hello world!";
```

6 Types

A type constrains the value of an expression to adhere to a particular space of values. There are five fundamental types in Typescript-on-LLVM, namely: `boolean`, `number`, `string`, `void`, and `Array`.

The type-specifiers are:

type-specifier:

```
  boolean
  boolean[]
  number
  number[]
  string
  string[]
  void
```

These types are defined below.

6.1 `boolean`

A value has `boolean` type if it is either `true` or `false`.

6.2 `number`

All numbers in Typescript-on-LLVM are floating-point values consisting of decimal literals, represented as a sequence of digits, optionally followed by a single '.' and a trailing sequence of digits. Hexadecimal values are not supported.

Examples of numbers are `1`, `3.14`, etc.

6.3 `string`

A String is a sequence of one or more characters enclosed in double-quotes, e.g. "foo". In order to include a double-quote in a string constant, the escape character slash ('\') may be used. To include a slash in a string, use another slash, e.g. "\\x" may be used to represent the literal string "\x".

Examples of strings are: `"x"`, `"foo"`, `"foo bar baz"`, etc.

6.5 `void`

The `void` type is used to indicate the absence of a value being returned from a function.

For example:

```
function sleep(): void {
    /* ... */
}
```

6.4 `Array`

An `Array` is an indexable collection of values of the same type.

It may be declared using `let` or `const`, as described in Section 5.

For example, to declare an array of numbers:

```
let values: number[] = [1, 2, 3];
```

References

In preparing this LRM and the scanner/parser code, I consulted the following:

- Kernighan & Ritchie, The C Programming Language, Second Edition, 1988.
- Parsing with ocamllex and Menhir
- Menhir Reference Manual
- ocamlyacc Tutorial
- The MicroC Compiler code on the class website (referenced in Piazza)
- ANSI C yacc Grammar